Our Ref. No. 042390.P8104
Express Mail No.: EL466333336US

UNITED STATES PATENT APPLICATION

FOR

PROTECTING SOFTWARE ENVIRONMENT

IN ISOLATED EXECUTION

INVENTORS:

Carl M. Ellison
Roger A. Golliver
Howard C. Herbert
Derrick C. Lin
Francis X. McKeen
Gil Neiger
Ken Reneris
James A. Sutton
Shreekant S. Thakkar
Millind Mittal

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Blvd., 7th Floor
Los Angeles, CA 90025-1026
(714) 557-3800

BACKGROUND

1. Field of the Invention

This invention relates to microprocessors. In particular, the invention relates to processor security.

2. Description of Related Art

Advances in microprocessor and communication technologies have opened up many opportunities for applications that go beyond the traditional ways of doing business. Electronic commerce (E-commerce) and business-to-business (B2B) transactions are now becoming popular, reaching the global markets at a fast rate. Unfortunately, while modern microprocessor systems provide users convenient and efficient methods of doing business, communicating and transacting, they are also vulnerable to unscrupulous attacks. Examples of these attacks include virus, intrusion, security breach, and tampering, to name a few. Computer security, therefore, is becoming more and more important to protect the integrity of the computer systems and increase the trust of users.

Threats caused by unscrupulous attacks may be in a number of forms. Attacks may be remote without requiring physical accesses. An invasive remote-launched attack by hackers may disrupt the normal operation of a system connected to thousands or even millions of users. A virus program may corrupt code and/or data of a single-user platform.

Existing techniques to protect against attacks have a number of drawbacks. Anti-virus programs can only scan and detect known viruses. Most anti-virus programs use a weak policy in which a file or program is assumed good until proved bad. For many

security applications, this weak policy may not be appropriate. In addition, most anti-virus programs are used locally where they are resident in the platform. This may not be suitable in a group work environment. Security co-processors or smart cards using cryptographic or other security techniques have limitations in speed performance, memory capacity, and flexibility. Redesigning operating systems creates software compatibility issues and causes tremendous investment in development efforts.

BRIEF DESCRIPTION OF THE DRAWINGS

The features and advantages of the present invention will become apparent from the following detailed description of the present invention in which:

Figure 1A is a diagram illustrating a logical operating architecture according to one embodiment of the invention.

Figure 1B is a diagram illustrating accessibility of various elements in the operating system and the processor according to one embodiment of the invention.

Figure 1C is a diagram illustrating a computer system in which one embodiment of the invention can be practiced.

Figure 2 is a diagram illustrating a secure platform according to one embodiment of the invention.

Figure 3A is a diagram illustrating a subset of a software environment having a usage protector according to one embodiment of the invention.

Figure 3B is a diagram illustrating a subset of a software environment having a usage protector according to another embodiment of the invention.

Figure 3C is a diagram illustrating the subset of a software environment according to yet another embodiment of the invention.

Figure 3D is a diagram illustrating the subset of a software environment according to yet another embodiment of the invention.

Figure 3E is a diagram illustrating the subset of a software environment according to yet another embodiment of the invention.

Figure 4 is a flowchart illustrating a process to protect usage of a subset of a software environment according to one embodiment of the invention.

Figure 5 is a flowchart illustrating the process to protect usage of the subset according to another embodiment of the invention.

Figure 6 is a flowchart illustrating the process to protect usage of the subset according to yet another embodiment of the invention.

Figure 7 is a flowchart illustrating a process to protect usage of the subset according to yet another embodiment of the invention.

DETAILED DESCRIPTION

In the following description, for purposes of explanation, numerous details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that these specific details are not required in order to practice the present invention. In other instances, well-known electrical structures and circuits are shown in block diagram form in order not to obscure the present invention.

ARCHITECTURE OVERVIEW

One principle for providing security in a computer system or platform is the concept of an isolated execution architecture. The isolated execution architecture includes logical and physical definitions of hardware and software components that interact directly or indirectly with an operating system of the computer system or platform. An operating system and the processor may have several levels of hierarchy, referred to as rings, corresponding to various operational modes. A ring is a logical division of hardware and software components that are designed to perform dedicated tasks within the operating system. The division is typically based on the degree or level of privilege, namely, the ability to make changes to the platform. For example, a ring-0 is the innermost ring, being at the highest level of the hierarchy. Ring-0 encompasses the most critical, privileged components. In addition, modules in Ring-0 can also access to lesser privileged data, but not vice versa. Ring-3 is the outermost ring, being at the lowest level of the hierarchy. Ring-3 typically encompasses users or applications level and has the least privilege. Ring-1 and ring-2 represent the intermediate rings with decreasing levels of privilege.

Figure 1A is a diagram illustrating a logical operating architecture 50 according to one embodiment of the invention. The logical operating architecture 50 is an abstraction

of the components of an operating system and the processor. The logical operating architecture 50 includes ring-0 10, ring-1 20, ring-2 30, ring-3 40, and a processor nub loader 52. The processor nub loader 52 is an instance of an processor executive (PE) handler. The PE handler is used to handle and/or manage a processor executive (PE) as will be discussed later. The logical operating architecture 50 has two modes of operation: normal execution mode and isolated execution mode. Each ring in the logical operating architecture 50 can operate in both modes. The processor nub loader 52 operates only in the isolated execution mode.

Ring-0 10 includes two portions: a normal execution Ring-0 11 and an isolated execution Ring-0 15. The normal execution Ring-0 11 includes software modules that are critical for the operating system, usually referred to as kernel. These software modules include primary operating system (e.g., kernel) 12, software drivers 13, and hardware drivers 14. The isolated execution Ring-0 15 includes an operating system (OS) nub 16 and a processor nub 18. The OS nub 16 and the processor nub 18 are instances of an OS executive (OSE) and processor executive (PE), respectively. The OSE and the PE are part of executive entities that operate in a secure environment associated with the isolated area 70 and the isolated execution mode. The processor nub loader 52 is a protected bootstrap loader code held within a chipset in the system and is responsible for loading the processor nub 18 from the processor or chipset into an isolated area as will be explained later.

Similarly, ring-1 20, ring-2 30, and ring-3 40 include normal execution ring-1 21, ring-2 31, ring-3 41, and isolated execution ring-1 25, ring-2 35, and ring-3 45, respectively. In particular, normal execution ring-3 includes N applications 42_1 to 42_N and isolated execution ring-3 includes K applets 46_1 to 46_K.

One concept of the isolated execution architecture is the creation of an isolated region in the system memory, referred to as an isolated area, which is protected by both the processor and chipset in the computer system. The isolated region may also be in cache memory, protected by a translation look aside (TLB) access check. Access to this isolated region is permitted only from a front side bus (FSB) of the processor, using special bus (e.g., memory read and write) cycles, referred to as isolated read and write cycles. The special bus cycles are also used for snooping. The isolated read and write cycles are issued by the processor executing in an isolated execution mode. The isolated execution mode is initialized using a privileged instruction in the processor, combined with the processor nub loader 52. The processor nub loader 52 verifies and loads a ring-0 nub software module (e.g., processor nub 18) into the isolated area. The processor nub 18 provides hardware-related services for the isolated execution.

One task of the processor nub 18 is to verify and load the ring-0 OS nub 16 into the isolated area, and to generate the root of a key hierarchy unique to a combination of the platform, the processor nub 18, and the operating system nub 16. The operating system nub 16 provides links to services in the primary OS 12 (e.g., the unprotected segments of the operating system), provides page management within the isolated area, and has the responsibility for loading ring-3 application modules 45, including applets 46_1 to 46_K, into protected pages allocated in the isolated area. The operating system nub 16 may also load ring-0 supporting modules.

The operating system nub 16 may choose to support paging of data between the isolated area and ordinary (e.g., non-isolated) memory. If so, then the operating system nub 16 is also responsible for encrypting and hashing the isolated area pages before evicting the page to the ordinary memory, and for checking the page contents upon restoration of the page. The isolated mode applets 46_1 to 46_K and their data are tamper-resistant and monitor-resistant from all software attacks from other applets, as well as

from non-isolated-space applications (e.g., 42_1 to 42_N), dynamic link libraries (DLLs), drivers and even the primary operating system 12. Only the processor nub 18 or the operating system nub 16 can interfere with or monitor the applet's execution.

Figure 1B is a diagram illustrating accessibility of various elements in the operating system 10 and the processor according to one embodiment of the invention. For illustration purposes, only elements of ring-0 10 and ring-3 40 are shown. The various elements in the logical operating architecture 50 access an accessible physical memory 60 according to their ring hierarchy and the execution mode.

The accessible physical memory 60 includes an isolated area 70 and a non-isolated area 80. The isolated area 70 includes applet pages 72 and nub pages 74. The non-isolated area 80 includes application pages 82 and operating system pages 84. The isolated area 70 is accessible only to elements of the operating system and processor operating in isolated execution mode. The non-isolated area 80 is accessible to all elements of the ring-0 operating system and to the processor.

The normal execution ring-0 11 including the primary OS 12, the software drivers 13, and the hardware drivers 14, can access both the OS pages 84 and the application pages 82. The normal execution ring-3, including applications 42_1 to 42_N, can access only to the application pages 82. Both the normal execution ring-0 11 and ring-3 41, however, cannot access the isolated area 70.

The isolated execution ring-0 15, including the OS nub 16 and the processor nub 18, can access to both of the isolated area 70, including the applet pages 72 and the nub pages 74, and the non-isolated area 80, including the application pages 82 and the OS pages 84. The isolated execution ring-3 45, including applets 46_1 to 46_K, can access only to the application pages 82 and the applet pages 72. The applets 46_1 to 46_K reside in the isolated area 70.

Figure 1C is a diagram illustrating a computer system 100 in which one embodiment of the invention can be practiced. The computer system 100 includes a processor 110, a host bus 120, a memory controller hub (MCH) 130, a system memory 140, an input/output controller hub (ICH) 150, a non-volatile memory, or system flash, 160, a mass storage device 170, input/output devices 175, a token bus 180, a motherboard (MB) token 182, a reader 184, and a token 186. The MCH 130 may be integrated into a chipset that integrates multiple functionalities such as the isolated execution mode, host-to-peripheral bus interface, memory control. Similarly, the ICH 150 may also be integrated into a chipset together or separate from the MCH 130 to perform I/O functions. For clarity, not all the peripheral buses are shown. It is contemplated that the system 100 may also include peripheral buses such as Peripheral Component Interconnect (PCI), accelerated graphics port (AGP), Industry Standard Architecture (ISA) bus, and Universal Serial Bus (USB), etc.

The processor 110 represents a central processing unit of any type of architecture, such as complex instruction set computers (CISC), reduced instruction set computers (RISC), very long instruction word (VLIW), or hybrid architecture. In one embodiment, the processor 110 is compatible with an Intel Architecture (IA) processor, such as the Pentium™ series, the IA-32™ and the IA-64™. The processor 110 includes a normal execution mode 112 and an isolated execution circuit 115. The normal execution mode 112 is the mode in which the processor 110 operates in a non-secure environment, or a normal environment without the security features provided by the isolated execution mode. The isolated execution circuit 115 provides a mechanism to allow the processor 110 to operate in an isolated execution mode. The isolated execution circuit 115 provides hardware and software support for the isolated execution mode. This support includes configuration for isolated execution, definition of an isolated area, definition (e.g.,

decoding and execution) of isolated instructions, generation of isolated access bus cycles, and generation of isolated mode interrupts.

In one embodiment, the computer system 100 can be a single processor system, such as a desktop computer, which has only one main central processing unit, e.g. processor 110. In other embodiments, the computer system 100 can include multiple processors, e.g. processors 110, 110a, 110b, etc., as shown in Figure 1C. Thus, the computer system 100 can be a multi-processor computer system having any number of processors. For example, the multi-processor computer system 100 can operate as part of a server or workstation environment. The basic description and operation of processor 110 will be discussed in detail below. It will be appreciated by those skilled in the art that the basic description and operation of processor 110 applies to the other processors 110a and 110b, shown in Figure 1C, as well as any number of other processors that may be utilized in the multi-processor computer system 100 according to one embodiment of the present invention.

The processor 110 may also have multiple logical processors. A logical processor, sometimes referred to as a thread, is a functional unit within a physical processor having an architectural state and physical resources allocated according to some partitioning policy. Within the context of the present invention, the terms "thread" and "logical processor" are used to mean the same thing. A multi-threaded processor is a processor having multiple threads or multiple logical processors. A multi-processor system (e.g., the system comprising the processors 110, 110a, and 110b) may have multiple multi-threaded processors.

The host bus 120 provides interface signals to allow the processor 110 or processors 110, 100a, and 110b to communicate with other processors or devices, e.g., the MCH 130. In addition to normal mode, the host bus 120 provides an isolated access

bus mode with corresponding interface signals for memory read and write cycles when the processor 110 is configured in the isolated execution mode. The isolated access bus mode is asserted on memory accesses initiated while the processor 110 is in the isolated execution mode. The isolated access bus mode is also asserted on instruction pre-fetch and cache write-back cycles if the address is within the isolated area address range and the processor 110 is initialized in the isolated execution mode. The processor 110 responds to snoop cycles to a cached address within the isolated area address range if the isolated access bus cycle is asserted and the processor 110 is initialized into the isolated execution mode.

The MCH 130 provides control and configuration of memory and input/output devices such as the system memory 140 and the ICH 150. The MCH 130 provides interface circuits to recognize and service isolated access assertions on memory reference bus cycles, including isolated memory read and write cycles. In addition, the MCH 130 has memory range registers (e.g., base and length registers) to represent the isolated area in the system memory 140. Once configured, the MCH 130 aborts any access to the isolated area that does not have the isolated access bus mode asserted.

The system memory 140 stores system code and data. The system memory 140 is typically implemented with dynamic random access memory (DRAM) or static random access memory (SRAM). The system memory 140 includes the accessible physical memory 60 (shown in Figure 1B). The accessible physical memory includes a loaded operating system 142, the isolated area 70 (shown in Figure 1B), and an isolated control and status space 148. The loaded operating system 142 is the portion of the operating system that is loaded into the system memory 140. The loaded OS 142 is typically loaded from a mass storage device via some boot code in a boot storage such as a boot read only memory (ROM). The isolated area 70, as shown in Figure 1B, is the memory area that is defined by the processor 110 when operating in the isolated execution mode.

Access to the isolated area 70 is restricted and is enforced by the processor 110 and/or the MCH 130 or other chipset that integrates the isolated area functionalities. The isolated control and status space 148 is an input/output (I/O)-like, independent address space defined by the processor 110 and/or the MCH 130. The isolated control and status space 148 contains mainly the isolated execution control and status registers. The isolated control and status space 148 does not overlap any existing address space and is accessed using the isolated bus cycles. The system memory 140 may also include other programs or data which are not shown.

The ICH 150 represents a known single point in the system having the isolated execution functionality. For clarity, only one ICH 150 is shown. The system 100 may have many ICH's similar to the ICH 150. When there are multiple ICH's, a designated ICH is selected to control the isolated area configuration and status. In one embodiment, this selection is performed by an external strapping pin. As is known by one skilled in the art, other methods of selecting can be used, including using programmable configuring registers. The ICH 150 has a number of functionalities that are designed to support the isolated execution mode in addition to the traditional I/O functions. In particular, the ICH 150 includes an isolated bus cycle interface 152, the processor nub loader 52 (shown in Figure 1A), a digest memory 154, a cryptographic key storage 155, an isolated execution logical processor manager 156, and a token bus interface 159.

The isolated bus cycle interface 152 includes circuitry to interface to the isolated bus cycle signals to recognize and service isolated bus cycles, such as the isolated read and write bus cycles. The processor nub loader 52, as shown in Figure 1A, includes a processor nub loader code and its digest (e.g., hash) value. The processor nub loader 52 is invoked by execution of an appropriate isolated instruction (e.g., Iso_Init) and is transferred to the isolated area 70. From the isolated area 80, the processor nub loader 52 copies the processor nub 18 from the system flash memory (e.g., the processor nub code

18 in non-volatile memory 160) into the isolated area 70, verifies and logs its integrity, and manages a symmetric key used to protect the processor nub's secrets. In one embodiment, the processor nub loader 52 is implemented in read only memory (ROM). For security purposes, the processor nub loader 52 is unchanging, tamper-resistant and non-substitutable. The digest memory 154, typically implemented in RAM, stores the digest (e.g., hash) values of the loaded processor nub 18, the operating system nub 16, and any other critical modules (e.g., ring-0 modules) loaded into the isolated execution space. The cryptographic key storage 155 holds a symmetric encryption/decryption key that is unique for the platform of the system 100. In one embodiment, the cryptographic key storage 155 includes internal fuses that are programmed at manufacturing. Alternatively, the cryptographic key storage 155 may also be created with a random number generator and a strap of a pin. The isolated execution logical processor manager 156 manages the operation of logical processors operating in isolated execution mode. In one embodiment, the isolated execution logical processor manager 156 includes a logical processor count register that tracks the number of logical processors participating in the isolated execution mode. The token bus interface 159 interfaces to the token bus 180. A combination of the processor nub loader digest, the processor nub digest, the operating system nub digest, and optionally additional digests, represents the overall isolated execution digest, referred to as isolated digest. The isolated digest is a fingerprint identifying the ring-0 code controlling the isolated execution configuration and operation. The isolated digest is used to attest or prove the state of the current isolated execution.

The non-volatile memory 160 stores non-volatile information. Typically, the non-volatile memory 160 is implemented in flash memory. The non-volatile memory 160 includes the processor nub 18. The processor nub 18 provides the initial set-up and low-level management of the isolated area 70 (in the system memory 140), including verification, loading, and logging of the operating system nub 16, and the management of

the symmetric key used to protect the operating system nub's secrets. The processor nub 18 may also provide application programming interface (API) abstractions to low-level security services provided by other hardware. The processor nub 18 may also be distributed by the original equipment manufacturer (OEM) or operating system vendor (OSV) via a boot disk.

The mass storage device 170 stores archive information such as code (e.g., processor nub 18), programs, files, data, applications (e.g., applications 42_1 to 42_N), applets (e.g., applets 46_1 to 46_K) and operating systems. The mass storage device 170 may include compact disk (CD) ROM 172, floppy diskettes 174, and hard drive 176, and any other magnetic or optical storage devices. The mass storage device 170 provides a mechanism to read machine-readable media. When implemented in software, the elements of the present invention are the code segments to perform the necessary tasks. The program or code segments can be stored in a processor readable medium or transmitted by a computer data signal embodied in a carrier wave, or a signal modulated by a carrier, over a transmission medium. The "processor readable medium" may include any medium that can store or transfer information. Examples of the processor readable medium include an electronic circuit, a semiconductor memory device, a ROM, a flash memory, an erasable programmable ROM (EPROM), a floppy diskette, a compact disk CD-ROM, an optical disk, a hard disk, a fiber optical medium, a radio frequency (RF) link, etc. The computer data signal may include any signal that can propagate over a transmission medium such as electronic network channels, optical fibers, air, electromagnetic, RF links, etc. The code segments may be downloaded via computer networks such as the Internet, an Intranet, etc.

I/O devices 175 may include any I/O devices to perform I/O functions. Examples of I/O devices 175 include a controller for input devices (e.g., keyboard, mouse,

trackball, pointing device), media card (e.g., audio, video, graphics), a network card, and any other peripheral controllers.

The token bus 180 provides an interface between the ICH 150 and various tokens in the system. A token is a device that performs dedicated input/output functions with security functionalities. A token has characteristics similar to a smart card, including at least one reserved-purpose public/private key pair and the ability to sign data with the private key. Examples of tokens connected to the token bus 180 include a motherboard token 182, a token reader 184, and other portable tokens 186 (e.g., smart card). The token bus interface 159 in the ICH 150 connects through the token bus 180 to the ICH 150 and ensures that when commanded to prove the state of the isolated execution, the corresponding token (e.g., the motherboard token 182, the token 186) signs only valid isolated digest information. For purposes of security, the token should be connected to the digest memory.

PROTECTING SOFTWARE ENVIRONMENT IN ISOLATED EXECUTION

The overall architecture discussed above provides a basic insight into a hierarchical executive architecture to manage a secure platform. The elements shown in Figures 1A, 1B, and 1C are instances of an abstract model of this hierarchical executive architecture. The implementation of this hierarchical executive architecture is a combination of hardware and software. In what follows, the processor executive, the processor executive handler, and the operating system executive are abstract models of the processor nub 18, the processor nub loader 52, and the operating system nub 16 (Figures 1A, 1B, and 1C), respectively.

Figure 2 is a diagram illustrating a secure platform 200 according to one embodiment of the invention. The secure platform 200 includes the OS nub 16, the processor nub 18, a key generator 240, a hashing function 220, and a usage protector 250,

all operating within the isolated execution environment, as well as a software environment 210 that may exist either inside or outside the isolated execution environment.

The OS nub or OS executive (OSE) 16 is part of the operating system running on the secure platform 200. The OS nub 16 has an associated OS nub identifier (ID) 201, that may be delivered with the OS nub 16 or derived from the OS nub code or associated information. The OS nub ID 201 may be a pre-determined code that identifies the particular version of the OS nub 16. It may also represent a family of various versions of the OS nub 16. The OS nub 16 may optionally have access to a public and private key pair unique for the platform 200. The key pair may be generated and stored at the time of manufacturing, at first system boot, or later. The private key 204 may be programmed into the fuses of a cryptographic key storage 155 of the input/output control hub (ICH) 150 or elsewhere in persistent storage within the platform 200. The private key 204 may be based upon a random number generated by an external random number generator. In one embodiment, the private key 204 is generated by the platform 200 itself the first time the platform 200 is powered up. The platform 200 includes a random number generator to create random numbers. When the platform 200 is first powered up, a random number is generated upon which the private key 204 is based. The private key 204 can then be stored in the multiple key storage 164 of the non-volatile flash memory160. The feature of the private key 204 is that it cannot be calculated from its associated public key 205. Only the OS nub 16 can retrieve and decrypt the encrypted private key for subsequent use. In the digital signature generation process, the private key 204 is used as an encryption key to encrypt a digest of a message producing a signature, and the public key 205 is used as a decryption key to decrypt the signature, revealing the digest value.

The processor nub 18 includes a master binding key (BK0) 202. The BK0 202 is generated at random when the processor nub 18 is first invoked, i.e., when it is first

executed on the secure platform 200. The key generator 240 generates a key operating system nub key (OSNK) 203 to be used by the usage protector 250. The key generator 240 receives the OS Nub identifier 201 and the BK0 202 to generate an OSNK 203. There are a number of ways for the key generator 240 to generate the OSNK 203. The key generator 240 generates the OSNK 203 by combining the BK0 202 and the OS Nub ID 201 using a cryptographic hash function. In one embodiment, the OS nub ID 201 identifies the OS nub 16 being installed into the secure platform 200. The OS nub ID 201 can be the hash of the OS nub 16, or a hash of a certificate that authenticates the OS nub 16, or an ID value extracted from a certificate that authenticates the OS nub 16. It is noted that a cryptographic hash function is a one-way function, mathematical or otherwise, which takes a variable-length input string, called a pre-image and converts it to a fixed-length, generally smaller, output string referred to as a hash value. The hash function is one-way in that it is difficult to generate a pre-image that matches the hash value of another pre-image. In one embodiment, the OS nub ID 201 is a hash value of one of the OS Nub 16 and a certificate representing the OS nub 16. Since the security of an algorithm rests in the key, it is important to choose a strong cryptographic process when generating a key. The software environment 210 may include a plurality of subsets (e.g., subset 230). The usage of the software environment 210 or the usage of the subset 230 is protected by the usage protector 250. The usage protector 250 uses the OSNK 203 to protect the usage of the subset 230. The software environment 210 may include an operating system (e.g., a Windows operating system, a Windows 95 operating system, a Windows 98 operating system, a Windows NT operating system, Windows 2000 operating system) or a data base. The subset 230 may be a registry in the Windows operating system or a subset of a database. Elements can be implemented in hardware or software.

The subset 230 is hashed by the hashing function 220 to produce a first hash value 206 and a second hash value 312. One way to detect intrusion or modification is to compare the state of the subset before and after a time period. The first and second hash values 206 and 312 are typically generated at different times and/or at different places.

The usage protector 250 is coupled to the key generator 240 to protect usage of the software environment 210 or the subset 230, using the OSNK 203. The usage protection includes protection against unauthorized reads, and detection of intrusion, tampering or unauthorized modification. If the two hash values are not the same, then the usage protector 250 knows that there is a change in the subset 230. If it is known that this change is authorized and an updated hash value has been provided, the usage protector 250 would merely report the result. Otherwise, the usage protector 250 may generate an error or a fault function. When a user is notified of the error, fault condition, he or she would know that the subset 230 has been tampered, modified. The user may take appropriate action. Therefore, the usage of the subset 230 is protected.

The are several different embodiments of the usage protector 250. In one embodiment, the usage protector 250 decrypts the subset using the OSNK 203. In two other embodiments, the usage protector 250 uses not only the OSNK 203 but also the first hash value 206 and the second hash value 312. Yet in two other embodiments, the usage protector uses the OSNK 203, the private key 204 and the pubic key 205.

Figure 3A is a diagram illustrating the usage protector 250 shown in Figure 2 according to one embodiment of the invention. The usage protector 250 includes a compressor 310 and an encryptor 360.

In one embodiment, the compressor 310 receives the subset 230 and compresses the subset 230 to generate a compressed subset. The encryptor 360 then encrypts the compressed subset using the OSNK 203. The OSNK 203 is provided to the usage

protection 250 by the key generator 240 as shown in Figure 2. The compressed subset is applied in the encryption process to save time (i.e. speed up) in the encryption process and/or space for storing the subset 230 in a memory. In another embodiment, the encryptor 360 takes the subset 230 without going through the compressing process and encrypts the subset 230 to generate an encrypted subset using the OSNK 203. The encrypting of the compressed subset or the subset 230 prevents unauthorized reads of the subset 230. To establish an authorized read, a request can be make to the OS nub and if the request is granted, the OSNK 203 is used to decrypt the encrypted or compressed encrypted subset 230.

Figure 3B is a block diagram of the usage protector 250 according to another embodiment of the invention. The invention includes a first encryptor 305, a decryptor 365, a storage medium 310, and a comparator 315.

The first encryptor 305 encrypts the first hash value 206 using the OSNK 203 to generate a first encrypted hash value 302. The encrypted first hash value 302 is then stored in the storage 310. Storage medium 310 may be any type of medium capable of storing the encrypted hash value 302. The storage medium 310 may be any type of disk, i.e., floppy disks, hard disks and optical disks) or any type of tape, i.e., tapes. The decryptor 365 decrypts the retrieved encrypted first hash value 303 using the OSNK 203. This decrypting process generates a decrypted hash value 366. This decrypted first hash value 366 is then compared to the second hash value 312 by the comparator 315 to detect if changes have been made in the subset 230. However, when the subset 230 is deliberately updated an authorized agent, the stored encrypted hash value is also updated. In the case where the modification of the subset is authorized, the second hash value 301 when compares with the updated decrypted hash 366, shows there is no detection of tampering or unauthorized change in the subset 230.

Figure 3C is a diagram illustrating the usage protector 250 shown in Figure 2 according to one embodiment of the invention. The usage protector 250 includes a decryptor 325, a signature generator 320, a storage medium 322, and a signature verifier 330.

The decryptor 325 accepts the OS nub's encrypted private key (i.e., protected) 204, and decrypts it using the OSNK 203, exposing the private key 328 for use in the isolated environment. The signature generator 320 generates a signature 304 for the subset 230 using the private key 328. It is noted that the subset 230 may be compressed before input it into the signature generator 320 to generate the signature 304. The signature algorithm used by the signature generator 320 may be public-key digital signature algorithms which makes use of a secret private key to generate the signature, and a public key to verify the signature. Example of algorithms include ElGama, Schnorr and Digital Signature Algorithms schemes just to name a few. In one embodiment, the generation of the signature 304 includes hashing the subset 230 to generate a before hash value, which is then encrypted using the private key 328 to generate the signature 304. The signature 304 is then saved in a storage medium 322. At a later time, the signature is retrieved from the storage 322, and the retrieved signature 306 is used, along with public key 205, by the signature verifier 330 to verify the subset 230. The signature verifier 330 verifies whether the subset 230 has been modified, producing a modified/not-modified indicator 331.. In one embodiment, the verification process includes decrypting the retrieved signature 306 using the public key 205 to expose the before hash value. The subset 230 is hashed to generate an after hash value. The before hash value is compared to the after hash value to detect whether the subset 230 has been modified. If the two hash values match, the subset 230 is the same as it was when the signature was generated.

Figure 3D is a block diagram of the usage protector 250 according to yet another embodiment of the invention. The usage protector 250 includes a decryptor 345, a

manifest generator 335, a signature generator 340, a storage medium 349, a signature verifier 350, and a manifest verifier 355.

The decryptor 345 accepts the OS nub's encrypted private key 204, and decrypts it using the OSNK 203, exposing the private key 348 for use in the isolated environment. The manifest generator 335 generates a manifest 307 for the subset 230. The manifest 307 represents the subset 230 in a concise manner. The manifest 307 may include a number of descriptors or entities, which characterize some relevant aspects of the subset 230. Typically, these relevant aspects are particular or specific to the subset 230 so that two different subsets have different manifests. In one embodiment, the manifest 307 represents a plurality of entities (i.e., a collection of entities) where each entry in the manifest 307 represents a hash (e.g., unique fingerprint) over one entity in the collection. The subset 230 is partitioned into one or more group where each group has a pointer and associated hash in the manifest 307. The manifest 307 is stored in a storage medium 349 for later use. The manifest 307 is also input to the signature generator 340 to generate a signature 308 over the manifest using the private key 348. The generated signature 308 is also stored in a storage medium 349. At a later time, we desire to verify that the portions of the subset 230 described by the manifest have not changed. This requires verifying that the manifest itself has not been changed, and that each group in subset 230 described by the manifest has not been changed. the stored signature and manifest are retrieved from the storage medium 349. The retrieved signature 309, and the retrieved manifest 354, along with the public key 205, are used by the signature verifier 350 to test that the retrieved manifest 354 is unchanged from the original manifest 307. The signature verifier 350 produces a signature-verifier flag 351, which is asserted when the signature verifies that the manifest is unchanged. In one embodiment, the verification process includes decrypting the retrieved signature 309 using the public key 205 to expose the before hash value. The retrieved manifest 354 is hashed to generate an after hash value.

The before hash value is compared to the after hash value to detect whether the retrieved manifest 354 has been modified. If the two hash values match, the retrieved manifest 354 is the same as it was when the signature was generated. The retrieved manifest 354 is also supplied to the manifest verifier 355, which uses the descriptive information in the retrieved manifest 354 to selectively verify portions of subset 230. In a typical embodiment, this involves hashing each group in subset 230, where the group is identified by information in the retrieved manifest 354, and comparing the newly generated hash value against the hash value for the group stored in the retrieved manifest 354. The manifest verifier 355 produces a manifest-verified flag 356, which is asserted if all entries described by the retrieved manifest 354 are verified as unchanged. If both the manifest-verified flag and the signature-verified flag are asserted, then the overall verification process passes, and selected portions of subset 230 are known to be the same as when the signed manifest was originally generated, and a pass/fail flag 358 is asserted. Note that the signature verifier 350 and the manifest verifier 355 can be invoked in any order.

Figure 3E is a diagram illustrating the usage protector 250 shown in Figure 2 according to another embodiment of the invention. The usage protector 250 includes a first encryptor 305, a second encryptor 365, a storage medium 310, and a comparator 315.

The first encryptor 305 encrypts the first hash value 206 using the OSNK 203. The first hash value 206 is provided by the hashing function 220 as shown in Figure 2. The first encryptor 305 takes the first hash value 206 and encrypts it to generate an encrypted first hash value 302 using the OSNK. The encrypted hash value 302 is then stored in a storage 310 for later use. The encryption by the OSNK 203 allows the encrypted hash value to be stored in arbitrary (i.e., unprotected) storage media. Storage medium 310 may be any type of medium capable of storing information (e.g., the

encrypted hash value 302). The storage medium 310 may be any type of disk, i.e., floppy disks, hard disks and optical disks) or any type of tape, i.e., tapes. The second encryptor 365 takes the second hash value 312 to encrypts it to generate an encrypted second hash value 301 using the OSNK 203. The second hash value 312 is provided by the hash function 220. The first encryptor 305 and the second encryptor 365 uses the same encryption algorithm, and this algorithm produces identical repeatable results for a given input. The encrypted first hash value 302 is now retrieved from the storage 310 for comparing with the second encrypted hash value 301. The comparator 315 compares the encrypted second hash value 301 with the retrieved encrypted first hash value 303 to detect if the subset 230 has been modified or tampered with. In the case where the subset 230 is deliberately updated by an authorized agent, the stored encrypted hash value is also updated. Since the modification of the subset 230 is authorized, the second encrypted hash value 301 is the same as the retrieved first encrypted first hash value 303. In the case where the subset 230 has been unauthorized modified or tampered with, the comparator 315 generates a modified/not-modified flag indicating the subset has been modified and therefore, the subset 230 should not be used. An attacker cannot simply replace the first encrypted hash value 303 with one corresponding to the unauthorized modified subset 230, because the attacker does not have access to the OSNK 203 used to encrypt the hash value.

Figure 4 is a flowchart illustrating a process 400 to protect usage of the software environment 210 or subset 230 according to one embodiment of the invention.

Upon START, the process 400 checks to see whether the subset needs protection (e.g., to prevent the subset from being read) (Block 401). If the subset does not need protection, the process 400 is terminated. If the subset wants protection, the process 400 then obtains the OSNK (Block 402) and the subset of the software environment (Block 403). Then the process 400 asks whether there is an authorized read of the subset (Block

403). If the read of the subset is not authorized, the process 400 encrypts the subset (Block 404). The process 400 is then terminated. If the read of the subset is authorized, the process 400 continues by decrypting the subset using the OSNK (Block 405). Next, the process 400 stores the decrypted subset in a storage (Block 406). The process 400 reads the subset by retrieving the decrypted subset from the storage (Block 407). Then the process 400 is terminated.

Figure 5 is a flowchart illustrating a process 500 to protect usage of the subset of the software environment according to one embodiment of the invention.

Upon START, the process 500 obtains the OSNK (Block 501).Then the process 500 obtains a hash value from the secure environment (Block 502). Next, the process 500 determines whether it is the first hash value (Block 503). If it is the first hash value, the process 500 encrypts the first hash value using the OSNK Block 504), The process 500 then stores the encrypted hash value in a storage for later use (Block 505). The process 500 is then terminated. If at Block 503 the hash value is not the first hash value, the process 500 retrieves the encrypted first hash value (Block 506). Then the process 500 decrypts the retrieved hash value (Block 507). The process 500 compares the two hash values (Block 508). The process 500 is terminated

Figure 6 is a flowchart illustrating a process to protect usage of a subset of a software environment according to yet another embodiment of the invention.

Upon START, the process 600 obtains the OSNK (Block 601). The process 600 then decrypts the protected private key, using the OSNK, to produce a private key (Block 602). The process 600 then generates a signature for the subset using the private key (Block 603). Next, the process 600 stores the signature into a storage (Block 604). The process 600 verifies the signature using the public key (Block 605). The verification

process detects whether the subset has been modified to generate a modified/not-modified flag (Block 606). The process 600 is then terminated.

Figure 7 is a flowchart illustrating a process 700 to protect usage of a subset of a software environment according to yet another embodiment of the invention.

Upon START, the process 700 obtains the OSNK and a hash value (Block 701). Next, the process 700 determines whether this is the first hash value (Block 702). If this is the first time the subset is encrypted, the obtained hash value is a first hash value. The process 700 encrypts the first hash value (Block 703) using the OSNK and stores the encrypted first hash value in a storage (Block 704). The process 800 is terminated. If, the process 700 determines that it is not the first hash value then the obtained hash value is a second hash value, the process 700 then encrypts the second hash value using the OSNK (Block 705). The process 700 then retrieves the encrypted first hash value from the storage (Block 706). Next, the process 700 compares the retrieved encrypted first hash value and the encrypted second hash value (Block 707). From the result of the comparison, the process 700 then generates a modified/not-modified flag after determine whether the subset has been modified. The process 700 is terminated.

While this invention has been described with reference to illustrative embodiments, this description is not intended to be construed in a limiting sense. Various modifications of the illustrative embodiments, as well as other embodiments of the invention, which are apparent to persons skilled in the art to which the invention pertains are deemed to lie within the spirit and scope of the invention.